A Comprehensive Introduction to Python Programming and GUI Design Using Tkinter

Bruno Dufour

McGill Univeristy SOCS

Contents

1 Overview of the Python Language **2**
- 1.1 Main Features . . . 2
- 1.2 Language Features . . . 2
  - 1.2.1 Literals . . . 2
  - 1.2.2 Variables . . . 4
  - 1.2.3 Operators . . . 5
  - 1.2.4 Data Structures . . . 6
  - 1.2.5 Control Structures and the `range( )` Function . . . 10
  - 1.2.6 Function Definitions and Function Calls . . . 12
  - 1.2.7 Classes . . . 13

2 Python and Tkinter for RAD and Prototyping **16**
- 2.1 Designing User Interfaces . . . 16
- 2.2 What is Tkinter? . . . 16
- 2.3 Why Tkinter? . . . 17
- 2.4 Nothing is perfect 18
- 2.5 Show me what you can do! . . . 18

3 Overview of Tkinter Widgets **24**
- 3.1 What is a Widget . . . 24
- 3.2 The Different Widgets . . . 24
  - 3.2.1 Toplevel . . . 24
  - 3.2.2 Button . . . 25
  - 3.2.3 Checkbutton . . . 25
  - 3.2.4 Entry . . . 26
  - 3.2.5 Frame . . . 26
  - 3.2.6 Label . . . 27
  - 3.2.7 Listbox . . . 27
  - 3.2.8 Menu . . . 28
  - 3.2.9 Message . . . 29
  - 3.2.10 OptionMenu . . . 29
  - 3.2.11 Radiobutton . . . 29
  - 3.2.12 Scale . . . 30
  - 3.2.13 Scrollbar . . . 32
  - 3.2.14 Text . . . 32
  - 3.2.15 Canvas . . . 32

3.3 Additional Widgets: the Python Mega Widgets (PMW) 37

4 Geometry Managers and Widget Placement **38**

4.1 What is Geometry Management? 38

4.2 The “Pack” Geometry Manager 38

4.3 The “Grid” Geometry Manager 40

4.4 The “Place” Geometry Manager 41

5 Tkinter-specific Topics **43**

5.1 Specifying colors 43

5.2 Specifying Fonts 43

5.3 Tkinter Variables 45

5.3.1 The Need for New Variables 45

5.3.2 Using Tkinter Variables 45

6 Event Handling: bringing applications to life **48**

6.1 The Idea 48

6.2 Event Types and Properties 49

6.3 Event Descriptors 50

6.4 Binding Callbacks to Events 52

6.5 Unit testing and Graphical User Interfaces: Simulating Events 53

A Tkinter color names **55**

1

Overview of the Python Language

1.1 Main Features

Python is an interpreted, interactive, object-oriented high-level language. Its syntax resembles pseudo-code, especially because of the fact that indentation is used to indentify blocks. Python is a dynamcally typed language, and does not require variables to be declared before they are used. Variables "appear" when they are first used and "disappear" when they are no longer needed.

Python is a scripting language like Tcl and Perl. Because of its interpreted nature, it is also often compared to Java. Unlike Java, Python does not require all instructions to reside inside classes.

Python is also a multi-platform language, since the Python interpreter is available for a large number of standard operating systems, including MacOS, UNIX, and Microsoft Windows. Python interpreters are usually written in C, and thus can be ported to almost any platform which has a C compiler.

1.2 Language Features

1.2.1 Literals

1.2.1.1 Integers

There are 3 kinds of integer literals:

Decimal integers : integers not starting with a '0' digit or the integer 0 (eg: 205).

Octal integers : integers which have a leading 0 as a prefix (eg: 0205).

Hexadecimal integers : integers which have 0x (or 0X) as a prefix (eg: 0x00FF0205).

Standard integers occupy 32 bits of memory, but Python allows appending ’L’ (or ’l’) to the integer to make it a long integer, which is unlimited in size (that is, as with all unlimited things in computer science, it is only limited by the amount of available memory).

1.2.1.2 Floats

There are two kinds of float literals:

Point Floats : floats containing a period followed by one or more digits (eg: 123.456).

Exponent Floats : floats in exponential form, ie which end with ’e’ (or ’E’) followed by a decimal integer exponent. Note that even if the integer part has a leading zero, it is still considered as decimal (eg: 123.456e78).

1.2.1.3 Strings

String constants in Python are enclosed either within two single quotes or within two double quotes, interchangeably.

Prefix	Description
u or U	Defines a Unicode string
r or R	Defines a Raw string (backslash characters are not interpreted as escape characters). Mostly used for regular expressions.

If a prefix must contain both a ’U’ and an ’R’, then the ’U’ must appear before the ’R’.

Standard C escape sequences are also accepted as part of the string constants, in addition to some escape sequences that are added by Python,as indicated below:

Sequence	Description
\newline	Ignored
\\	\
\'	Single quote
\"	Double quote
\a	ASCII Bell Character (BEL)
\b	ASCII Backspace Character (BS)
\f	ASCII Form Feed Character (FF)
\n	ASCII Line Feed Character (LF)
\Nname	Unicode character named 'name' (Unicode only)
\r	ASCII Carriage Return Character (CR)
\t	ASCII Horizontal Tab Character (TAB)
\uhhhh	Character with the 16-bit Hexadecimal value hhhh (Unicode Only)
\Uhhhhhhhh	Character with the 32-bit Hexadecimal value hhhhhhhh (Unicode Only)
\v	ASCII Vertical Tab Character (VT)
\ooo	ASCII Character with the octal value ooo
\hh	ASCII Character with the hexadecimal value hh

Notes:

1. Two string adjacent string literals will be concatenated at compile time. For example, `"Hello"" world!"` will be concatenated into `"Hello world!"`. This allows to easily break string literals across multiple lines. Comments are allowed between the parts of the string. Also, different quotation marks can be used for each part. The different parts of the string must be enclosed in parantheses if the string is to span multiple lines, if it is not already the case (eg for function calls).

2. String literals can also be enclosed within three double quotes, in which case newlines within the string are become part of it.

1.2.1.4 Imaginary Numbers

Python provides built-in support for imaginary numbers. It defines `1j` to be equal to $\sqrt{-1}$. Imaginary numbers can be written in the form `a + bj`, where `a` and `b` are integers or floats.

1.2.2 Variables

Variables are not declared in Python. Since Python is a dnamically typed language, any variable can be assigned any value, regardless of its previous type. This also holds for instance variables in classes. Therefore, it is a good idea to initialize all instance variables in the constructor to make sure that memory has been allocated for them, and in order to make sure that they can safely be used inside the class. Python used to rely on a reference counting algorithm to free the unused variables. The newer versions of Python (ie 2.0 beta and above) rely on a more traditional garbage collection technique.

This thus implies that allocated memory is automatically freed when it is no longer used.

It is usually very useful to think of Python variables as objects, since it is in essence how they are implemented.

1.2.2.1 Private Instance Variables

By default, all instance variables are declared as public variables. However, there is a possibly to use name mangling to make a variable “pseudo-private”. This is done by preceding the variable name with 2 or more underscores. In this case, the actual name of the variable will be a combination of the class name and the variable name, without the leading underscores removed from the class name and a single underscore being added as a prefix to the new name. For example, inside a class named ’my_class’, a variable named ’__my_var’ would actually be internally known as _my_class__my_var.

1.2.3 Operators

Python supports the standard C operators, along with some new operators, as described below.

1.2.3.1 Unary Operators

Operator	Description
+	Unary plus
+	Unary minus
∼	Bit inversion

1.2.3.2 Binary Operators

Operator	Description
	Arithmetic Operators
`+`	Binary addition
`-`	Binary subtraction
`*`	Binary multiplication
`/`	Binary division
`%`	Remainder of the integer division
`**`	Power
	Bitwise Operators
`<<, >>`	Bit shift
`&`	Bitwise AND
`|`	Bitwise OR
`^`	Bitwise XOR
	Logical Operators
`==`	Equal
`!=, <>`	Not equal
`<`	Greater than
`>`	Less than
`>=`	Greater than or equal
`<=`	Less than or equal
`is, is not`	Object identity
`in, not in`	Set membership
	Boolean Operators
`and`	Boolean AND
`or`	Boolean OR
`not`	Boolean NOT

Python allows combining multiple logical operators into a more intuitive form to avoid the use of the keyword `and`. For example, instead of writing `(x > 10 and x <= 20`, one could write `(10 < x <= 20)`.

1.2.4 Data Structures

The Python language provides built-in support for some very interesting data structures: lists, tuples and dictionaries. These data structures can be classified within two main categories: sequences and mappings. Sequences are finite ordered sets of elements. The elements of a sequence can be accessed by *indexing* using non-negative numbers. This indexing starts from 0 in Python. Mappings are also finite sets, but the way elements of a mapping are accessed differs. Mappings use arbitrary index sets for accessing the elements.

The Python language defines 3 main types of sequences: lists, tuples and strings (for simplicity, I gather under the term strings ordinary strings and unicode objects). At the current time, only one mapping type is available: the dictionary. Each of these is discussed in details below.

1.2.4.1 Lists

Lists are composed of an arbitrary number of elements. Each of these elements can have any valid type. Lists in Python are defined by enumerating the elements of a list, separated by commas, and enclosed within square brackets. Note that lists can be nested. For example:

```
[0, 1, 2, 4], ['arial', 20, 'bold'], [1, 'a string', [3, 4, 5]]
```

are valid lists.

To access an element of the list, it is necessary to know its index. The first element has index 0. Accessing elements using their index is done using square brackets, just like for an array reference in C or Java. For example:

Listing 1: Accessing List Elements

```
>> list = ['a', 'b', 'c']
>> list[0]
'a'
>> list[2]
'c'
>>['a', 'b', 'c'][1]
'b'
>>
```

The elements of a list can also be modified by assigning a value to a selection, just like it would be done for arrays in C or Java. For example:

Listing 2: Modifying lists

```
>> list = ['a', 'b', 'c']
>> list[1] = 'd'
>> list
['a', 'd', 'c']
>>
```

When a *negative* index is specified, the length of the list is added to it before it is used. This make a negative index “wrap” around. Therefore, -1 denotes the last element of the list, -2 the second to last element, and so on.

It is also possible to select multiple elements of a list at a time by taking a “slice” of the original list. Slicings are of the form `list[lower:upper]`. This creates a new sub-list containing the elements of the original list having an index k, $lower <= k < upper$. For example:

Listing 3: Slicing

```
>> list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> list[1:7]
[1, 2, 3, 4, 5, 6]
>>
```

It is possible to assign a list to a slice to modify part of an existing list, as shown below. Assigning an empty list to a slice deleted the slice from the original list.

Listing 4: Slice Assignment

```
>> list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> list[1:6] = ['a', 'b']
[0, 'a', 'b', 6, 7, 8, 9, 10]
>>
```

The colon operator has some interesting defaults. When its left-hand side is not specified, it defaults to 0. When its right-hand side is not specified, it defaults to the length of the list. Therefore, the Python instruction `list_name[:]` will return the entire list.

The built-in function `len( )` can be used to obtain the length of a list. It is also possible to concatenate lists by using the `'+'` operator.

1.2.4.2 Tuples

Tuples simply consist of a series of values enclosed within parentheses. Here is an example of a tuple:

('arial', 10, 'bold')

Tuples work exactly like lists, except for the fact that they are *immutable*, which means that their elements cannot be modified in place. However, the values stored in the components of a tuple *can* be modified. In other words, a tuple should be thought of as holding references (or pointers) to other objects. Consider the following two examples. The first one shows that trying to change the elements of a tuple is not allowed. The second one shows that it is possible to modify a mutable element which is inside of a tuple. Note the fundamental difference between the two examples.

Listing 5: Tuple Example 1

```
>> list = ['a', 'f', 'c']
>> tuple = (10, list, 'string constant')
>> tuple[1]
['a', 'f', 'c']
>> tuple[1] = ['some', 'other', 'list']
```

```
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Listing 6: Tuple Example 2

```
>> list = ['a', 'f', 'c']
>> tuple = (10, list, 'string constant')
>> tuple[1]
['a', 'f', 'c']
>> tuple[1][1] = 'b'
>> list
['a', 'b', 'c']
>> tuple
(10, ['a', 'b', 'c'], 'string constant')
>>
```

Tuples, just like lists, can be concatenated using the `'+'` operator.

1.2.4.3 Dictionaries

Dictionaries are mutable objects similar to lists, but are not restricted to integers for indexing purposes. For example, using dictionaries allows to *map* string values to objects, and reference these objects using their associated string. This is similar to what is done in a common english dictionary, where definitions are looked up by word. The index for a dictionary is called a *key*, and can be anything except a list, a dictionary, or other mutable types that are compared by value rather than by identity, since Python requires that keys have a constant hash value.

Dictionaries in Python are declared by listing an arbitrary number of key/datum pairs sperated by a comma, all enclosed within curly brackets. Key/datum pairs are of the form `key : datum`. Thus, a dictionary declaration has the following form:

```
{key0 : datum0, key1 : datum1, ..., keyN : datumN}
```

The square brackets are again used to access elements of a dictionary. However, this time, a key is used instead of an integer index.

Here is a Python example using dictionaries:

Listing 7: Dictionary Examples

```
>> dict = \{"alice":"123-4567", "bob":"234-5678", "charlie":"N/A"\}
>> dict['alice']
'123-4567'
>> dict['charlie']
'N/A'
>> dict['charlie'] = '987-6543'
>> dict
\{'alice': '123-4567', 'bob': '234-5678', 'charlie': '987-6543'\}
>>
```

1.2.5 Control Structures and the `range( )` Function

Python supports all of the tradional control structures. They are described in detail in this section.

1.2.5.1 If statement

The general syntax for an “if” statement is as follows:

Listing 8: If Statement Syntax

```
if(expr):
    statements
elif (expr):
    statements
elif (expr):
    statements
else:
    statements
```

Each condition is tested starting at the top-most `if` statement until a true condition is evaluated. If no condition is true, then the statements in the `else` block are evaluated (if present).

1.2.5.2 While statement

The general syntax for a “while” statement is as follows:

Listing 9: While Statement Syntax

```
while expr:
    statements
else:
    statements
```

The statements in the `while` block are executed as long as *`expr`* is true. When it evaluates to false (ie to 0), then the statements in the `else` block are executed (if present).

1.2.5.3 For statement

The general syntax for a “for” statement is as follows:

Listing 10: For Statement Syntax

```
for target in list:
    statements
else:
    statements
```

The statements in the `for` block are evaluated for each element of `list`. The current element of `list` an any given iteration is accessible in the `target` variable. Once all elements have been processed, the `else` clause is executed (if present).

Since for loops in Python are required to work on lists, the language provides the built-in function `range( )`, which is used to generate a list of numbers. Its has three major forms, each of which returns a list of integers.

- `range(upper)`: returns a list of consecutive integers x such that $0 <= x < upper$. For example

Listing 11:

```
>> range(5)
[0, 1, 2, 3, 4]
>>
```

- `range(lower, upper)`: returns a sorted list of consecutive integers x such that $lower <= x < upper$. For example

Listing 12:

```
>> range(2, 8)
[2, 3, 4, 5, 6, 7]
>> range(8, 2)
[]
>>
```

- `range(from, to, increment)`: returns a sorted list of integers starting from `from` and ending at `to` but separated by `increment`. Note that the value of `to` is not included in the bound. For example,

Listing 13:

```
>> range(8, 4, -1)
[8, 7, 6, 5]
>> range(1, 7, 2)
[1, 3, 5]
>>
```

1.2.6 Function Definitions and Function Calls

Function in Python are declared using the `def` keyword. The basic syntax is as follows:

Listing 14: Functions

```
def func_name(arg1, arg2, ..., argN):
    statements
```

Since the language is untyped, no function definition has a return type, and parameters are simply listed. If a function is to return a value, the keyword `return` can be used to return it. The same keyword used without a value will stop the excution of the function. This also the convention that C and Java use.

Python also supports some interesting features relative to function calls. The first of these features is that arguments can be passed and filled in two distinct ways: the traditional (ie C and Java) way, which is by position. Alternatively, Python supports filling in arguments by name. When arguments are provided by name, then their placement in the argument list does not matter, since the name has precedence over the place in the list. To make this concept clearer, here are two examples illustrating the two methods previously described:

Listing 15: Arguments Example 1 - filled in by position

```
>> def foo(w, x, y, z):
       return (w, x, y, z)
>> foo(1, 2, 3, 4)
(1, 2, 3, 4)
>>
```

Listing 16: Arguments Example 2 - filled in by name

```
>> def foo(w, x, y, z):
       return (w, x, y, z)
>> foo(x = 2, z = 4, w = 1, y = 3)
(1, 2, 3, 4)
>>
```

It is also possible to combine the two methods in the same function call, provided that all arguments following the first one that is passed by name are also passed by name. The arguments before the first one that is passed by name will all be passed by position.

Note that multiple definitions of a single argument will raise an exception. An exception is also raised if not all arguments receive a value, the arguments that are ignored have been assigned a default value, as explained next.

The second feature of the Python language relative to function calls is that arguments can receive a default value, in which case no value needs to be explicitely specified for the arguement. If a value is indeed specified, it replaces the default value for the argument. Default values are simply assigned to the arguments in the function declaration. Here is a very simple examples to illustrate this feature:

Listing 17: Mixed Argument Passing

```
>> def power(base, exp = 2):
     return x ** exp
>> power(2, 3)
8
>> power(5)
25
>>
```

The only restriction imposed by the language relative to default arguments is that all arguments following the first one that is assigned a default value must also be assigned a default value.

By combining default arguments and named arguments, it is easy to observe that function calls can be drastically reduced in length. This allows for faster prototyping and can simplify repetitive calls.

Python also allows functions to be declared using **arg* and ***arg* as arguments. An argument of the form **arg* will match any of the remaining non-named arguments (and will be a tuple), while ***arg* will match the remaining named arguments, and be of type Dictionary. This allows to define methods which can take an arbitrary number of parameters.

1.2.7 Classes

Python being an object-oriented language, it allows to create new object types in the form of classes. Declaring classes in Python is trivial. Here is the general syntax for a class declaration:

Listing 18: Class Declaration Syntax

```
class class_name inheritance_list:
     class_variables

     method0(arg0, arg1, ..., argN):
          statements

     method0(arg0, arg1, ..., argN):
          statements

     methodM(arg0, arg1, ..., argN):
          statements
```

The *inheritance_list* argument is optional. It must be included in brackets if present. It can contain one or more class names, separated by commas. It will most often contain only one class name: the base class of the current class, ie the class that it inherits from.

Python supports multiple inheritance in the form of *mixin* classes. These classes inherit from multiple base classes (ie the *inheritance_list* contains more than one class. Python uses a depth-first search, left-to-right strategy to find attributes in multiple inheritance cases. Whenever a list of base classes is specified, the leftmost class will be searched first for a specific attribute, followed by all of its base classes. If nothing is found in these classes, then the second base class will be searched in the same way, and so on.

Python has no access specifier (like private, public or protected in Java). By default, all methods declared within a class are public. Simulating private methods is still possible by using the same name mangling feature as it was done for variables (see section 1.2.2.1 on page 5). This is done by preceding the method name by two or more underscores.

Inside a class, the `self` keyword refers to the current instance of this class.

1.2.7.1 Special function names

Python defines some special function names that have a predefined meaning. These reserved names all have the form __*special_name*__, so as to minimize the possibility of name conflicts. The following table lists the most common ones and describes their meaning.

Method	Description
`__init__`	The class constructor. Use this method to assign values to instance variables in order to allocate space for them and thus avoid runtime errors that could occur if a use occurs before a definition. This method can take parameters if needed. If the base class defines an `__init__` method, call it using the form *baseclass_name*.`__init__(self`*[, args]*`)`. Constructors cannot return a value.
`__del__(self)`	The class destructor. If the base class defines this method, then it must be called to ensure proper destrcution of the instance.
`__str__(self)`	String representation of a class. It is the equivalent of the `toString( )` method in Java. It must return a string object describing the instance.
`__repr__(self)`	"Official" string representation of a class. This method should also return a string, but the description returned by this method should be more complete and useful, since it is usually used for debugging purposes. The Python Language Reference states that this representation should usually be a valid python statement that allows to recreate the instance of a class.
`__cmp__(self, other)`	Used to compare two instances of a class. Must return a negative integer if `self < other`, zero if `self = other` and a positive integer if `self > other`. NOTE: Python 2.1 introduces the `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__ and __ge__` methods, which allow to extend comparions to common operators. This is very similar to operator overloading in C++.
`__nonzero__(self)`	Truth value testing. Should return 0 or 1 only, depending on whether the instance should evaluate to false or not.

2

Python and Tkinter for RAD and Prototyping

2.1 Designing User Interfaces

User interfaces are what allows end users to interact with an application. An application can be excellent, but without a good user interface, it becomes more difficult to use, and less enjoyable. It is thus very important to design good user interfaces.

Designing user interface takes place at two different levels: the graphical level and the event level. Graphical elements of a user interface are called *widgets*. Widgets are basic components like buttons, scrollbars, etc. But user interfaces involve more than a collection of widgets placed in a window. The application must be able to respond to mouse clicks, keyboard actions or system events such as minimizing the window. For this to happen, events must be associated to some pieces of code. This process is called *binding*. The next two chapters will cover each level in more details, but this chapter will present an overview of Tkinter and explain why it has become the leading GUI toolkit for the Python language.

2.2 What is Tkinter?

Tkinter is an open source, portable graphical user interface (GUI) library designed for use in Python scripts. Tkinter relies on the Tk library, the GUI library used by Tcl/Tk and Perl, which is in turn implemented in C. Thus, Tkinter is implemented using multiple layers.

Several competing GUI toolkits are available to use with the Python language, namely:

wxPython : a wrapper extension for wxWindows, a portable GUI library originally developed for the C++ language. It is the second most popular GUI toolkit for Python since it is considered excellent for complex interface design.

JPython (Jython) : since it is implemented in java, JPython has access to Java GUI libraries, namely SWING and AWT. Recently, JTkinter has been implemented and provides a Tkinter port to JPython using the Java Native Interface (JNI).

PyKDE / PyQt, PyGTK : these packages provide an access to KDE and Gnome GUI libraries to python scripts.

Win32all.exe : provides access to Microsoft Foundation Classes (MFC) to python scripts. It is limited to run on MS Windows only.

WPY : a GUI library that can be used on both Microsoft Windows and UNIX X Windows. This library uses the MFC coding style.

X11 : a library based on the X Windows and Motif libraries allowing excellent control of the X11 environment, but are limited to run on X Windows OS's only.

2.3 Why Tkinter?

With all the competing GUI toolkits available for the Python language, what makes Tkinter stand out of the rest? Why is it the most popular toolkit for use interface design?

To find the answer, one must look at the advantages that it offers.

1. **Layered design** The layered approach used in designing Tkinter gives Tkinter all of the advantages of the TK library. Therefore, at the time of creation, Tkinter inherited from the benefits of a GUI toolkit that had been given time to mature. This makes early versions of Tkinter a lot more stable and reliable than if it had been rewritten from scratch. Moreover, the conversion from Tcl/Tk to Tkinter is really trivial, so that Tk programmers can learn to use Tkinter very easily.

2. **Accessibility** Learning Tkinter is very intuitive, and therefore quick and painless. The Tkinter implementation hides the detailed and complicated calls in simple, intuitive methods. This is a continuation of the Python way of thinking, since the language excels at quickly building prototypes. It is therefore expected that its preferred GUI library be implemented using the same approach. For example, here is the code for a typical "Hello world"-like application:

Listing 19: A Simple Application

```
from Tkinter import *
root = Tk( )
root.title('A simple application')
root.mainloop( )
```

The first 2 lines allow to create a complete window. Compared to MFC programming, it makes no doubt that Tkinter is simple to use. The third line sets the caption of the window, and the fourth one makes it enter its event loop.

3. **Portability** Python scripts that use Tkinter do not require modifications to be ported from one platform to the other. Tkinter is available for any platform that Python is implemented for, namely Microsoft Windows, X Windows, and Macintosh. This gives it a great advantage over most competing libraries, which are often restricted to one or two platforms. Moreover, Tkinter will provide the native look-and-feel of the specific platform it runs on.

4. **Availability** Tkinter is now included in any Python distribution. Therefore, no supplementary modules are required in order to run scripts using Tkinter.

2.4 Nothing is perfect . . .

Tkinter has many advantages that make it the primary choice for UI design. However, it has a drawback which originates directly from its imlementation: it has a significant overhead due to its layered implementation. While this could constitute a problem with older, slower machines, most modern computers are fast enough so as to cope with the extra processing in a reasonable amount of time. When speed is critical, however, proper care must be taken so as to write code that is as efficient as possible.

Even if the overhead tends to become less relevant with time, there is still a disadvantage to this layered implementation that is not going to fade away: the source code for the Tkinter library is only a shell that provides Tk's functionality to Python programs. Therefore, in order to understand what the Tkinter source code does, programmers need to understand the source code of the Tk library, which is in turn written in C. While certainly not impossible, this requires more work from programmers and can be quite time consuming.

2.5 Show me what you can do!

In order to fully understand the advantages that Tkinter has to offer, it is necessary to see some examples demonstrating just how simple programming using this toolkit can be. Let's start with a more detailed explanation of the example presented above. For simplicity, the listing of the complete source of the example is reproduced again below.

Listing 20: A very simple application

```
from Tkinter import *
root = Tk( )
root.title('A simple application')
root.mainloop( )
```

While this example is almost the smallest Tkinter program that can be written (the third line is optional, but was added to make the application slightly more interesting). Let's analyze the code line by line.

```
from Tkinter import *
```

This line imports the whole Tkinter library. It must present in every Tkinter program.

```
root = Tk()
```

This line creates a complete window, called the *Tk root widget*. There should be only one root widget per application, and it must be created before all other widgets. This does not mean that Tkinter applications cannot have multiple windows at the same time. More complex examples showing how to create multiple windows within a single application will be discussed later.

```
root.title('A simple application')
```

This line invokes a method of the Tk root widget instance to set its title to 'A simple application'. For the purposes of this example, this is the simplest way of displaying text in the window. The next example will use another widget to achieve a similar goal.

```
root.mainloop( )
```

The call to `mainloop` makes the application enter its *event loop*, ie it makes it able to respond to user events, such as mouse events and keyboard events.

The next example is very similar to the first example, but it uses new *widgets* (visual components) to display some text and exit the application. Widgets will be discussed in the next chapter.

Listing 21: Creating widgets

```
from Tkinter import *

def quit():
    import sys; sys.exit()

root = Tk( )
lbl = Label(root, text="Press the button below to exit")
lbl.pack()
btn = Button(root, text="Quit", command=quit)
btn.pack()
root.mainloop( )
```

This example introduces some new concepts. First of all, it creates new widgets to put on the root window. Also, it responds to a specific user event (a click on the "Quit" button) by exiting the program.

```
def quit():
    import sys; sys.exit()
```

These two lines are not Tkinter-specific. They simply create a function which terminates the application when invoked.

```
lbl = Label(root, text="Press the button below to exit")
```

This line creates a new instance of the `Label` class. The first argument of any widget constructor is always to *master* widget, in other words to widget which will hold the new widget. In this case, the Tk root widget is specified as master, so that the new widgets will appear in the window we have created. The following arguments are usually passed by keyword, and represent various options of the widget. The available options depend on the widget type. Widgets will be discussed in details in chapter 3. In this specific case, the only option that is needed is the text to be displayed by the widget. Widgets have their default value for all of the available options, so that they do not need to be specified unless the default value does not fit the needs of the programmer.

```
lbl.pack()
```

Even though a instance of the `Label` class was created by the previous line, it does not yet appear on the root window. A *geometry manager* needs to be used in order to place the new widget in its master widget. In this case, the packer does this in a very simple way. Its default behaviour is to pack every new widget in a column, starting at the top of the master. Geometry managers will be discussed in greater details in chapter 4.

```
btn = Button(root, text="Quit", command=quit)
btn.pack()
```

These two lines again create a new widget, but this time it is an instance of the Button class. This button will display the text “Quit”, and the `quit` function will be invoked when it is clicked. This packer is once again responsible for placing this widget on the master window.

```
root.mainloop()
```

Once all widgets have been created, `root.mainloop()` is called in order to enter the event loop and allow the widgets to receive and react to user events (in this case a click on the button).

For the next example, we will rewrite Example 2 (above) to make it into a class. This is usually how more complex Tkinter programs are implemented, since it is a much cleaner way to organize things.

Listing 22: An object-oriented approach

```
from Tkinter import *

class Example3:
    def __init__(self, master):
        self.lbl = Label(master, text="Press the button below to exit")
        self.lbl.pack()
        self.btn = Button(master, text="Quit", command=self.quit)
        self.btn.pack()
```

```
    def quit(self):
        import sys; sys.exit()

root = Tk( )
ex3 = Example3(root)
root.mainloop( )
```

This example defines a new class: `Example3`. This class only has a constructor and a `quit` method. The `quit` method is almost identical to the `quit` function from Example 2, except that it must take a parameter, `self`, to receive a reference to the class instance when invoked. For more details, see section 1.2.7 on page 13.

For the last example in this section, let's build a larger interface. This fourth example will include many widgets and more complex widget placement. Do not worry about the implementation details for now, since everything will be covered in the subsequent sections. The code for this example is as follows:

Listing 23: A more complex example

```
from Tkinter import *

class Example4:
    def __init__(self, master):
        # showInfo needs to know the master
        self.master = master
        # Create a frame to hold the widgets
        frame = Frame(master)

        # Create a Label to display a header
        self.headLbl = Label(frame, text="Widget Demo Program", relief=RIDGE)
        self.headLbl.pack(side=TOP, fill=X)

        # Create a border using a dummy frame with a large border width
        spacerFrame = Frame(frame, borderwidth=10)

        # Create another frame to hold the center part of the form
        centerFrame = Frame(spacerFrame)
        leftColumn = Frame(centerFrame, relief=GROOVE, borderwidth=2)
        rightColumn = Frame(centerFrame, relief=GROOVE, borderwidth=2)

        # Create some colorful widgets
        self.colorLabel = Label(rightColumn, text="Select a color")
        self.colorLabel.pack(expand=YES, fill=X)

        entryText = StringVar(master)
        entryText.set("Select a color")
        self.colorEntry = Entry(rightColumn, textvariable=entryText)
        self.colorEntry.pack(expand=YES, fill=X)

        # Create some Radiobuttons
        self.radioBtns = []
        self.radioVal = StringVar(master)
        btnList = ("black", "red", "green", "blue", "white", "yellow")
        for color in btnList:
            self.radioBtns.append(Radiobutton(leftColumn, text=color, value=color, indicatoron=TRUE,
```

```
                                        variable=self.radioVal, command=self.updateColor))
        else:
            if (len(btnList) > 0):
                self.radioVal.set(btnList[0])
                self.updateColor()

        for btn in self.radioBtns:
            btn.pack(anchor=W)

        # Make the frames visible
        leftColumn.pack(side=LEFT, expand=YES, fill=Y)
        rightColumn.pack(side=LEFT, expand=YES, fill=BOTH)
        centerFrame.pack(side=TOP, expand=YES, fill=BOTH)

        # Create the Indicator Checkbutton
        self.indicVal = BooleanVar(master)
        self.indicVal.set(TRUE)
        self.updateIndic()
        Checkbutton(spacerFrame, text="Show Indicator", command=self.updateIndic,
                variable=self.indicVal).pack(side=TOP, fill=X)

        # Create the Color Preview Checkbutton
        self.colorprevVal = BooleanVar(master)
        self.colorprevVal.set(FALSE)
        self.updateColorPrev()
        Checkbutton(spacerFrame, text="ColorPreview", command=self.updateColorPrev,
                variable=self.colorprevVal).pack(side=TOP, fill=X)

        # Create the Info Button
        Button(spacerFrame, text="Info", command=self.showInfo).pack(side=TOP, fill=X)

        # Create the Quit Button
        Button(spacerFrame, text="Quit!", command=self.quit).pack(side=TOP, fill=X)

        spacerFrame.pack(side=TOP, expand=YES, fill=BOTH)
        frame.pack(expand=YES, fill=BOTH)

    def quit(self):
        import sys; sys.exit()

    def updateColor(self):
        self.colorLabel.configure(fg=self.radioVal.get())
        self.colorEntry.configure(fg=self.radioVal.get())

    def updateIndic(self):
        for btn in self.radioBtns:
            btn.configure(indicatoron=self.indicVal.get())

    def updateColorPrev(self):
        if (self.colorprevVal.get()):
            for btn in self.radioBtns:
                color = btn.cget("text")
                btn.configure(fg=color)
        else:
            for btn in self.radioBtns:
                btn.configure(fg="black")
```

```
    def showInfo(self):
        toplevel = Toplevel(self.master, bg="white")
        toplevel.transient(self.master)
        toplevel.title("Program info")
        Label(toplevel, text="A simple Tkinter demo", fg="navy", bg="white").pack(pady=20)
        Label(toplevel, text="Written by Bruno Dufour", bg="white").pack()
        Label(toplevel, text="http://www.cs.mcgill.ca/~bdufou1/", bg="white").pack()
        Button(toplevel, text="Close", command=toplevel.withdraw).pack(pady=30)


root = Tk()
ex4 = Example4(root)
root.title('A simple widget demo')
root.mainloop()
```

This example may look long at first, but it is only about 100 lines long, including comments and blank lines. Considering the fact that it creates two windows and numerous widgets, and that it responds to user events, the source is on fact relatively short. It could have been written in a yet shorter form, but this would have made the source more difficult to understand. An interesting experiment innvolves running this program and trying to resize its window. Widgets modify their size to accomodate the new window size as much as possible. This behaviour is a demonstration of Tkinter's geometry managers at work.

Finally, this application demonstrates how easy widget configuration is in Tkinter. The Checkbuttons even reconfigure widgets on the fly, using only a few lines of code. Note how the window is resized when the indicator option for the Radiobuttons is turned on and off.

3
Overview of Tkinter Widgets

3.1 What is a Widget

According to the Free On-Line Dictionary of Computing, the definition of a widget is:

> *widget*: [possibly evoking ”window gadget”] In graphical user interfaces, a combination of a graphic symbol and some program code to perform a specific function. E.g. a scroll-bar or button. Windowing systems usually provide widget libraries containing commonly used widgets drawn in a certain style and with consistent behaviour.

A widget is therefore a graphical object that is available from the Tkinter library. It is a kind of graphical building block. Intuitively, widgets are implemented as classes in Tkinter. Each widget therefore has a constructor, a destructor, its own set of properties and methods, and so on. While most other GUI toolkits have a very complex widget hierarchy, Tkinter's hierarchy is extremely simple. All widgets (like Button, Checkbutton, etc.) are dereived from the Widget class. All widget subclasses occupy the same level in the hierachy tree.

3.2 The Different Widgets

3.2.1 Toplevel

The Toplevel is technically not a widget, although this is more or less transparent to the user.

3.2.2 Button

The Button widget is a rectangular widget that is able to display text. This text can occupy several lines (if the text itself contains newlines or if wrapping occurs). Moreover, this text can have an underlined character (usually indicating a keyboard shortcut). Buttons are usually used to execute an action when they are clicked, so they offer the `command` option which associates a callback with the event corresponding to a mouse click with the left mouse button on the widget.

The Button widget provides the following options:

Option	Description
default	Specifies one of three states (DISABLED, NORMAL, ACTIVE) which determines if the button should be drawn with the style used for default buttons.

The Button widget provides the following methods:

Method	Description
flash()	Redraws the button several times, alternating between active and normal appearance.
invoke()	Invokes the callback associated with the button widget.

3.2.3 Checkbutton

The Checkbutton displays some text (or image) along with a small square called indicator. The indicator is drawn differently depending on the state of the checkbutton (selected or not). The Checkbutton widget supports the following option:

Option	Description
indicatoron	Specifies wheter the indicator should be drawn or not. If false, the Checkbutton acts like a toggle button.
offvalue	Specifies the value to store in the variable associated with the Checkbutton when it is not selected.
onvalue	Specifies the value to store in the variable associated with the Checkbutton when it is selected.
selectcolor	Specifies the background color to use for the widget whenever it is selected. This color applies to the indicator only if `indicatoron` is TRUE. It applies to the whole widget when this option is turned off.
variable	Specifies the name of a Tkinter variable that is to be assigned the values specified by the `onvalue` and `offvalue` options.

The Checkbutton widget defines the following methods:

Method	Description
deselect()	Deselects the checkbutton and stores the value specified by the offvalue option to its associated variable.
flash()	Redraws the button several times, alternating between active and normal colors.
invoke()	Simulates a click on the widget (toggles the state, updates the variable and invokes the callback associated with the checkbutton, if any).
select()	Selects the checkbutton and stores the value specified by the onvalue option to its associated variable.
toggle()	Similar to invoke(), but does not invoke the callback associated with the checkbutton.

3.2.4 Entry

The Entry widget allows users to type and edit a single line of text. A portion of this text can be selected.

The Entry widget provides the following options:

Option	Description
show	Controls how to display the contents of the widget. If non-empty, the widget will replace characters to be displayed by the first character the specified string. To get a password entry widget, use ”*”.

The Entry widget defines the following methods:

Method	Description
delete(index), delete(from,to)	Delete the character at index, or within the given range. Use delete(0, END) to delete all text in the widget.
get()	Returns the current contents of the widget.
insert(index, text)	Inserts text at the given index. Use insert(INSERT, text) to insert text at the cursor, insert(END, text) to append text to the widget.

3.2.5 Frame

The Frame widget is a rectangular container for other widgets. It is primarily used as a gemometry master since it allows to group widgets. It can have an unvisible border which is useful to space widgets. It can also have a visible border with a 3D effect, or no border at all.

The Frame widget provides the following options:

Option	Description
cursor	Specifies the mouse cursor to be used when the cursor is within the frame.
takefocus	Specifies whether the user should be able to move the focus to this widget by using the tabulation key (Tab).
width, height	Specifies the size of the widget.

The Frame widget does not provide any methods except the standard Widget methods.

3.2.6 Label

The Label widget is used to display text. It can only display text in a single font at a time. The text can span more than one line. In addition, one character in the text can be underlined, usually indicating a keyboard shortcut.

The Label widget provides the following options:

Option	Description
anchor	Specifies how the text should be placed in a widget. Use one of N, NE, E, SE, S, SW, W, NW, W, NW, CENTER.
bitmap	Specifies the bitmap to be displayed by the widget. If the image option is given, this option is ignored.
image	Specifies the image to be displayed by the widget. If specified, this takes precedence over the text and bitmap options.
justify	Defines how to align multiple lines of text relative to each others. Use LEFT, RIGHT, or CENTER. Differs from anchor in that justify specifies how to align multiple lines of text relative to each other.
text	Specifies the text to be displayed by the widget.
textvariable	Specifies the name of a variable that contains the text to be displayed by the widget. If the contents of the variable is modified, the widget will be updated accordingly.

The Label widget does not provide any methods except the standard Widget methods.

3.2.7 Listbox

The Listbox widget displays a list of strings. Each string is displayed on a separate line. It allows to insert, modify or delete strings from the list. The listbox can only contain text items, and all items must have the same font and colour. Depending on the widget configuration, the user can choose one or more alternatives from the list.

The Listbox widget provides the following options:

Option	Description
selectmode	Specifies the selection mode. One of SINGLE, BROWSE, MULTIPLE, or EXTENDED. Default is BROWSE. Use MULTIPLE to get checklist behavior, EXTENDED if the user usually selects one item, but sometimes would like to select one or more ranges of items.
xscrollcommand, yscrollcommand	Used to connect a listbox to a scrollbar. These options should be set to the `set` methods of the corresponding scrollbars.

The Listbox widget defines the following methods:

Method	Description
delete(index), delete(first, last)	Deletes one or more items. Use delete(0, END) to delete all items in the list.
get(index)	Gets one or more items from the list. This function returns the string corresponding to the given index (or the strings in the given index range). Use get(0, END) to get a list of all items in the list. Use ACTIVE to get the selected (active) item(s).
insert(index, items)	Inserts one or more items at given index (index 0 is before the first item). Use END to append items to the list. Use ACTIVE to insert items before the selected (active) item.
size( )	Returns the number of items in the list.

3.2.8 Menu

The Menu widget is used to implement toplevel, pulldown, and popup menus.

The Menu widget provides the following options:

Option	Description
postcommand	If specified, this callback is called whenever Tkinter is about to display this menu. If you have dynamic menus, use this callback to update their contents.
tearoff	If set, menu entry 0 will be a “tearoff entry”, which is usually a dashed separator line. If the user selects this entry, Tkinter creates a small Toplevel with a copy of this menu.
tearoffcommand	If specified, this callback is called when this menu is teared off.
title	Specifies the title of menu.

The Menu widget defines the following methods:

Method	Description
add(type, options...)	Appends an entry of the given type to the menu. The type argument can be one of "command", "cascade" (submenu), "checkbutton", "radiobutton", or "separator".
insert(index, type, options...)	Same as add and friends, but inserts the new item at the given index.
entryconfig(index, options...)	Reconfigures the given menu entry. Only the given options are changed.
delete(index)	Deletes one or more menu entries.

3.2.9 Message

The Message widget is used to display multiple lines of text. It is very similar to a plain label, but can adjust its width to maintain a given aspect ratio.

The Message widget provides the following option:

Option	Description
aspect	Specifies a non-negative integer describing the aspect ratio of the widget. This integer is taken to be the value of 100 * width / height. Default is 150, i.e. width:height = 1.5:1.

The Label widget does not provide any methods except the standard Widget methods.

3.2.10 OptionMenu

OptionMenu inherits from Menubutton. It is used to display a drop-down list of options. It only defines a constructor of the form:

```
OptionMenu(master, variable, value, *values)
```

where master is the *master* widget of this OptionMenu, *variable* is a Tkinter Variable (see section 5.3 on page 45), *value* and **values* are the values displayed by the OptionMenu. The documentation states that *value* will be the default variable, but it is not the case. The only point in having a separate *value* argument is to ensure that at least one value is to be displayed. To set the default value (or any value at any point of the execution), use `variable.set(value)`.

3.2.11 Radiobutton

The Radiobutton widget used to implement one-of-many selections. Radiobuttons can contain text or images, and you can associate a callback with each button to be excuted when the button is pressed. Each group of Radiobutton widgets should be associated with a single variable. Each button then represents a single value for that variable.

The Radiobutton widget provides the following options:

Option	Description
indicatoron	Specifies whether the indicator should be drawn or not. If false, the Radiobutton behaves like a toggle Button.
selectcolor	Specifies a background color to use when the widget is selected. If "indicatoron" is true, then the color applies to the indicator, otherwise it applies to the background of the widget.
value	Specifies the value to store in the variable when this button is selected (see "variable" below)
variable	Specifies a Tkinter Variable (see section 5.3 on page 45) that holds the value for the currently selected Radiobutton

The Radiobutton widget defines the following methods:

Method	Description
select()	Selects the Radiobutton and stores its value in its Tkinter variable.
deselect()	Deselects the Radiobutton. The value of its associated variable cannot be used (variable.get() will throw an exeception).
invoke()	Simulates a mouse click on the Radiobutton. Same as select() but also executes the callback, if there is one.

3.2.12 Scale

The Scale widget is used to display a slider which allows the user to select a value within a specified range.

The Scale widget provides the following options:

Option	Description
digits	Specifies the number of significant digits used when converting a value to a string. If negative, Tkinter ensures that every possible slider position will be converted to a different string while using the minimum number of digits.
from_, to	Specifies the range of values for the Scale
label	Specifies a string to be displayed by the Scale.
length	Specifies the width or height of the Scale, in screen pixels, depending on its orientation.
orient	Defines the orientation of the scale. One of HORIZONTAL or VERTICAL.
resolution	If greater than zero, then all Scale values will be rounded to multiples of this value. If less than zero, no rouding occurs. Default is 1 (scale takes integer values only).
showvalue	Specifies whether the scale should display its current value. Default is TRUE.
sliderlength	Specifies the length of the slider, in screen units. Default is 30.
sliderrelier	Specifies the relief used to draw the slider.
tickinterval	Specifies the spacing between tick marks displayed by the Scale. If it is 0, no tick marks are displayed.
troughcolor	Specifies the color used to fill the trough area.
variable	Specifies a Tkinter variable to hold the value of the scale.

The Scale widget defines the following methods:

Method	Description
coords(value=None)	Returns the coordinates of a point along the center of the trough that corresponds to *value*. The current value of the scale is used if *value* is not specified.
get()	Gets the current scale value. Tkinter returns an integer if possible, otherwise a floating point value is returned.
identity(x, y)	Returns a string describing which part of the Scale is situated at the location (x, y). TROUGH1 indicates a position above or on the left of the slider, TROUGH2 a position down or right of the slider, SLIDER a position on the SLIDER, and an empty string a position outside the Scale.
set()	Sets the scale value.

3.2.13 Scrollbar

The Scrollbar widget is a typical scrollbar, with arrows at both ends and a slider portion in the middle. Unlike typical scrollbars, its color can be modified.

The Scrollbar widget provides the following options:

Option	Description
command	Used to update the associated widget. This is typically the xview or yview method of the scrolled widget. If the user drags the scrollbar slider, the command is called as callback(MOVETO, offset) where offset 0.0 means that the slider is in its topmost (or leftmost) position, and offset 1.0 means that it is in its bottommost (or rightmost) position. If the user clicks the arrow buttons, or clicks in the trough, the command is called as callback(SCROLL, step, what). The second argument is either ”-1” or ”1” depending on the direction, and the third argument is UNITS to scroll lines (or other units relevant for the scrolled widget), or PAGES to scroll full pages.
orient	Defines how to draw the scrollbar. Use one of HORIZONTAL or VERTICAL. Default is VERTICAL.

The Scrollbar widget defines the following methods:

Method	Description

3.2.14 Text

3.2.15 Canvas

The canvas widget provides the basic graphics facilities for Tkinter, and so more advanced functions. Drawing on the canvas is done by creating various items on it. It is important to note at this stage that items are *not* widgets, even though they are similar in many ways. Each item on a canvas is enclosed by a bounding box, which is defined using 2 points: the top-left corner and the bottom-right corner of the box. Tkinter uses two coordinate systems simultaneously: the canvas coordinate system and the window coordinate system. Both systems express positions relative to the top-left corner, with X-coordinates increasing to the right and Y-coordinates increasing downwards. However, the origin for the two systems is different. The window system expresses the coordinates by placing the origin at the top-left corner of the visible portion of the canvas, while the canvas system places its origin at the top-corner of the canvas widget, even it is not visible. The difference is important when handling mouse events bound to the canvas since the event object receives its coordinates in the window system. Luckily, the canvas widget provides the necessary methods to convert coordinates to the canvas system, through calls to the canvasx( ) and canvasy( ) methods.

The Tkinter canvas supports the following standard items (more can also be added):

- Arc: arc, chord, pieslice
- Bitmap: builtin or read from an XBM file
- Image: a BitmapImage or PhotoImage instance
- Line
- Oval: circle or ellipse
- Polygon
- Resctangle
- Text
- Window: used to place other widgets on the canvas (makes the canvas widget act like a geometry manager)

Apart from their bounding box, all canvas items can be referenced using their unique item ID, which is assigned at the time they are created, and is returned by any of the item constructors. Tkinter also provides a usefull way of working with items: the tags. A number of strings, called tags, can be associated with a canvas item, and more than one item can have the same tag. Items can then be referenced using those tags (see section 3.2.15.1 on page 36 for details). Canvas items can also be bound to events, using the tag_bind( ) method.

The Canvas widget provides the following options:

Option	Description
closeenough	Specifies a float defining how close to an item the mouse cursor has to be before it is considered to be over it (a higher value means that the item will selected more readily). Default is 1.0.
confine	If TRUE (default), it is not allowed to set the canvas view outside of the region defined by “scrollregion” (see below).
scrollregion	Defines the region that is considered to be the boundary of all items in the canvas. It is used for scrolling purposes, in order to limit the scroll actions to a definite area.
xscrollcommand, yscrollcommand	Specifies the function used to interact with a scrollbar.
xscrollincrement, yscrollincrement	Specifies the increment for horizontal and vertical scrolling, respectively.

The Canvas widget defines the following methods:

Method	**Description**
create_arc(bbox, options)	Creates an *arc* canvas item and returns its item ID.
create_bitmap(position, options)	Creates a *bitmap* canvas item and returns its item ID.
create_image(position, options)	Creates an *image* canvas item and returns its item ID.
create_line(coords, options)	Creates a *line* canvas item and returns its item ID.
create_oval(bbox, options)	Creates an *oval* canvas item and returns its item ID.
create_polygon(coords, options)	Creates a *polygon* canvas item and returns its item ID.
create_rectangle(coords, options)	Creates a *rectangle* canvas item and returns its item ID.
create_text(position, options)	Creates a *text* canvas item and returns its item ID.
create_window(position, options)	Places a Tkinter widget in a canvas window item and returns its item ID.
delete(items)	Deletes all matching items, if any.
itemcget(item, option)	Returns the current value for an option from a canvas item.
itemconfig(item, options), itemconfigure(item, options)	Modifies one or more options for all matching items.
coords(item)	Returns a tuple containing the coordinates for the item.
coords(items, x0, y0, x1, y1, ..., xn, yn)	Changes the coordinates for all matching items.
bbox(items), bbox( )	Returns the bounding box for the given items. If the specifier is omitted, the bounding box for all items are returned.
canvasx(screenx), canvasy(screeny)	Converts a window coordinate to a canvas coordinate.
tag_bind(item, sequence, callback), tag_bind(item, sequence, callback, `"+"`)	Adds an event binding to all matching items. Using the “+” option, it adds the binding to the previous ones, otherwise all previous bindings are replaced.
tag_unbind(item, sequence)	Removes the binding, if any, for the given event sequence on all the matching items.
type(item)	Returns the type of the given item as a string (one of “arc”, “bitmap”, “image”, “line”, “oval”, “polygon”, “rectangle”, “text”, “window”).
lift(item), tkraise(item) / lower(item)	Moves the given item to the top / bottom of the canvas stack. If multiple items match, they are all moved while preserving their relative order.

Continued on next page...

Canvas Widget Methods - Continued

Method	Description
move(item, dx, dy)	Moves all matching items dx canvas units to the right, and dy canvas units downwards. Negative coordinates specify a displacement in the other direction.
cale(item, xscale, yscale, xoffset, yoffset)	Scales matching items according to the given scale factors. The coordinates for each item are first moved by -offset, then multiplied with the scale factor, and then moved back again.
addtag_above(newtag, item)	Adds newtag to the item just above the given item in the stacking order.
addtag_all(newtag)	Adds newtag to all items on the canvas. This is a shortcut for addtag_withtag(newtag, ALL).
addtag_below(newtag, item)	Adds newtag to the item just below the given item, in the stacking order.
addtag_closest(newtag, x, y)	Adds newtag to the item closest to the given coordinate.
addtag_enclosed(newtag, x1, y1, x2, y2)	Adds newtag to all items *completely* enclosed by the given rectangle.
addtag_overlapping(newtag, x1, y1, x2, y2)	Adds newtag to all items enclosed by or touching the given rectangle.
addtag_withtag(newtag, tag)	Adds newtag to all items having the given tag.
dtag(item, tag)	Removes the given tag from all matching items. If the tag is omitted, all tags are removed from the matching items. It is not an error to give a specifier that doesn't match any items.
gettags(item)	Returns all tags associated with the item, in a tuple.
find_above(tag, item)	Finds the item just above the given item in the stacking order.
find_all(tag)	Finds all items on the canvas. This is a shortcut for find_withtag(tag, ALL).
find_below(tag, item)	Finds the item just below the given item, in the stacking order.
find_closest(tag, x, y)	Finds the item closest to the given coordinate.
find_enclosed(tag, x1, y1, x2, y2)	Finds all items *completely* enclosed by the given rectangle.
find_overlapping(tag, x1, y1, x2, y2)	Finds all items enclosed by or touching the given rectangle.
find_withtag(tag, tag)	Finds all items having the given tag.
postscript(options)	Generates a postscript representation of the canvas items. Images and widgets are not included in the output.

Just like widgets, canvas items also have a number of options. All items can, for example, have their fill color and their border color changed, or set to transparent. The complete list can be found at http://www.python.org/.

3.2.15.1 Tagging in Tkinter

Tags represent a very powerful, yet very easy to use feature of Tkinter. The idea behind them is really simple, and it is not surprising to see them used in the implementation of sereval widgets, such as the Canvas and the Text widgets. In the Text widget, tags can be used to mark parts of the text for easier reference. In the canvas widget, the tags serve to mark on or more widgets for future reference. This discussion will focus on the use of tags in the Canvas widget.

As was previously mentioned, tags can be used in a many:many relationship: multiple canvas items can have the same tag and one item can have multiple different tags. This can be used to create groups of widgets. The most interesting point to make about tags is that canvas methods acting on items can receive as parameters item IDs or tags interchangeably. One might wonder what happens to methods that only apply to a single item at a time (such as bbox( )). Tkinter provides the best possible approach for these situations by only taking the first item that matches the specified tag. When the method must be applied to several or all of the items having the tag, Tkinter also provides an easy solution. The find_withtag( ) method produces a list of all items having the tag passed as only parameter. This information can then be used inside a simple for loop.

In addition to the tags that are added by the user, Tkinter supports two special, built-in tags, CURRENT and ALL. As expected, the ALL tag identifies all items in the canvas, while the CURRENT tag is automatically updated, indicating the item over which the mouse pointer is located. Therefore, a call to canvas.find_all( ) is the same as a call to find_withtag(ALL). Those two tags are simply handled like any other tag. The CURRENT constant simply evaluates to the string “current”, while the ALL constant evaluates to “all”. Therefore, those tags should not be used by the user. Doing so will not raise any exception, but setting these tags manually will silently fail since they are managed by Tkinter.

Moreover, the tag_bind( ) and tag_unbind( ) methods can be used to associate event callbacks to canvas items, thus freeing the canvas from the task of dispatching the event to the proper item(s). These methods reduce the gap between canvas items and widgets.

3.3 Additional Widgets: the Python Mega Widgets (PMW)

While Tkinter provides a good set of widgets to work with, it does not offer a complete set yet. For example, there is no combo box (an Entry widget combined with a drop-down list) widget in Tkinter. Luckily enough, there is another package that is compatible with Tkinter and that provides the elements that are missing: the Python Mega Widgets. It is available from http://pmw.sourceforge.net/.

4

Geometry Managers and Widget Placement

4.1 What is Geometry Management?

Geometry management consists of placing widget placement and size on the screen. Tkinter provides three geometry managers, therefore allowing the user to quickly and efficiently build user interfaces, with maximal control over the disposition of the widgets. The geometry managers also allow the user make abstraction of the specific implementation of the GUI on each platform that Tkinter supports, thus making the design truly portable.

Geometry management implies a great deal of negotiations between the different widgets and their containers, as well as the windows they are placed in. Geometry management is not only responsible for the precise position and size of a widget, but also of this widget's position and size relative to the other widgets, and of renegotiating these factors when conditions change, for example when the window is resized.

The subsequent sections will discuss each of the geometry managers in details.

4.2 The "Pack" Geometry Manager

The packer is the quickest way to design user interfaces using Tkinter. It allows the user the place the widgets relative to their contained widget. It is the most commonly used geometry manager since it allows a fair amount of flexibility.

The packer positions the slave widgets on the master widget (container) from the edges to the center, each time using the space left in the master widget by previous packing operations. The options that can be used in the `pack()` method are:

Option	**Values**	**Effect**
expand	YES (1), NO(0)	Specifies whether the widget should expand to fill the available space (at packing time and on window resize)
fill	NONE, X, Y, BOTH	Specifies how the slave should be resized to fill the available space (if the slave is smaller than the available space)
side	TOP (default), BOTTOM, LEFT, RIGHT	Specifies which side of the master should be used for the slave.
in_('in')	Widget	Packs the slaves inside the widget passed as a parameter. This option is usually left out, in which case the widgets are packed in their parent. *Restriction*: widgets can only be packed inside their parents or descendants of their parent.
padx,pady	Integer values	Specifies the space that must be left out between two adjacent widgets.
ipadx, ipady	Integer values	Specifies the size of the optional internal border left out when packing.
anchor	N, S, W, E, NW, SW, NE, SE, NS, EW, NSEW, CENTER (default)	Specifies where the widget should be placed in the space that it has been allocated by the packer, if this space is greater than the widget size.

The "Pack" geometry manager defines the following methods for working with widgets:

Method	**Effect**
pack(option=value,...) pack_configure(option=value,...)	Packs the widget with the specified options.
pack_forget()	The widget is no longer managed by the Pack manager, but is not destroyed.
pack_info()	Returns a dictionary containing the current options for the widget.
pack_slaves()	Returns a list of widget IDs, in the packing order, which are slaves of the master widget.

4.3 The “Grid” Geometry Manager

The grid geometry manager is used for more complex layouts. It allows the user to virtually divide the master widget into several rows and columns, which can then be used to place slave widgets more precisely. The pack geometry manager would require the use of multiple nested frames to obtain the same effect.

The grid geometry manager allows the user to build a widget grid using an approach that is very similar to building tables using HTML. The user builds the table specifying not only the row and column at which to place the widget, but also the rwo span and column span values to use for the widget. In addition to that, the sticky option can allow almost any placement of the widget inside a cell space (if it is bigger than the widget itself). Combinations of values for the sticky option also allow to resize the widget, such as EW value, equivalent to an expand option combined with a fill=X for the packer. The options that can be used in the `grid()` method are:

Option	Values	Effect
row, column	Integer values	Specifies where the widget should be positioned in the master grid.
rowspan, columnspan	Integer values	Specifies the number of rows / columns the widget must span across.
in_('in')	Widget	Uses the widget passed as a parameter as the master. This option is usually left out, in which case the widgets are packed in their parent. *Restriction*: widgets can only be packed inside their parents or descendants of their parent.
padx,pady	Integer values	Specifies the space that must be left out between two adjacent widgets.
ipadx, ipady	Integer values	Specifies the size of the optional internal border left out when packing.
sticky	N, S, W, E, NW, SW, NE, SE, NS, EW, NSEW	Specifies where the widget should be placed in the space that it has been allocated by the grid manager, if this space is greater than the widget size. Default is to center the widget, but the `grid()` method does not support the CENTER value for sticky.

The “Grid” geometry manager defines the following methods for working with widgets:

Method	Effect
grid(option=value,...), grid_configure(option=value,...)	Places the widget in a grid, using the specified options.
grid_forget(), grid_remove()	The widget is no longer managed by the Grid manager, but is not destroyed.
grid_info()	Returns a dictionary containing the current options.
grid_slaves()	Returns a list of widget IDs which are slaves of the master widget.
grid_location(x,y)	Returns a (column, row) tuple which represents the cell in the grid that is closest to the point (x, y).
grid_size()	Returns the size of the grid, in the form of a (column, row) tuple, in which column is the index of the first empty column and row the index of the first empty row.

It is important to note that empty rows and columns are not displayed by the grid geometry manager, even if a minimum size is specified. Note: The grid manager cannot be used in combination with the pack manager, as this results in an infinite negociation loop.

4.4 The “Place” Geometry Manager

The place geometry manager is the most powerful manager of all since it allows exact placement of the widgets inside a master widget (container). However, it is more difficult to use and usually represents a great amount of overhead that is rarely needed.

The Place geometry manager allows placement of widgets using either exact coordinates (with the x and y options), or as a percentage relative to the size of the master window (expressed as a float in the range [0.0, 1.0]) with the relx and rely options. The same principle holds for the widget size (using width / height and/or relwidth / relheight). The options that can be used in the `place()` method are:

Option	Values	Effect
anchor	N, NE, E, SE, SW, W, NW (Default), CENTER	Specifies which part of the widget should be placed at the specfied position.
bordermode	INSIDE, OUTSIDE	Specifies if the outside border should be taken into consideration when placing the widget.
in_('in')	Widget	Places the slave in the master passed as the value for this option.
relwidth, relheight	Float [0.0, 1.0]	Size of the slave widget, relative to the size of the master.
relx, rely	Float [0.0, 1.0]	Relative position of the slave widget.
width, hieght	Integer values	Absolute width and height of the slave widget.
x, y	Integer Values	Absolute position of the slave widget.

The “Place” geometry manager defines the following methods for working with widgets:

Method	Effect
place(option=value,. . .) place_configure(option=value,. . .)	Places the widget with the specified options.
place_forget()	The widget is no longer managed by the Place manager, but is not destroyed.
place_info()	Returns a dictionary containing the current options for the widget.
place_slaves()	Returns a list of widget IDs which are slaves of the master widget.

5
Tkinter-specific Topics

5.1 Specifying colors

Tkinter defines several color names that can be easily used as color values in Python programs. These color names are taken from the names that are provided by X Windows. The complete list is prodided in Appendix A on page 55. For a more precise description of colors, Tkinter allows the use of *color strings*. These strings start with a number sign (#) and specify the precise values of the red, green and blue components of the color. There are three forms of color strings, depending on how many levels each component is allowed:

#RGB : 4-bit components (12 bits total), 16 levels per component.

#RRGGBB : 8-bit components (24 bits total), 256 levels pre component.

#RRRRGGGGBBBB : 16-bit components, (48 bits total), 65536 levels per component.

R, G and B are single hexadecimal digits (0..F, or 0..f).

For example, "#F00", "#FF0000" and "#FFFF00000000" all specify a pure red color. Similarly, "#48A", "#4080A0" and "#40008000A000" represent the same color.

5.2 Specifying Fonts

Tkinter provides several ways of defining fonts. It provides font descriptors that are easier to work with than the very precise but too confusing X Window System font descriptor. X Window System font descriptors are still supported, but they will not be discussed here since they should be rarely needed. There are two other major kinds of font descriptors that are available: in Tkinter:

Tuples or Strings : Font can be decribed using an n-tuple of the form:

(*family*, *size*, [*option1*], [*option2*], . . .).

The font family is in fact its name. It is a string describing the font. For example, “arial”, “courier”, or “times new roman” are all valid font families. The size is given as an integer. The remaining options are facultative, and there can be as many of them as required. The valid options are: normal, roman, bold, italic, underline and overstrike.

It is also possible to specify a font using a single string if the font name does not contain any spaces. For example, (“arial”, 10, “italic”, “bold”) is equivalent to “arial 10 italic bold”.

Font instances : This requires to import a separate module, named `tkFont`. This modules defines the `Font` class. An instance of this class can be used anywhere a standard Tkinter font descriptor is valid. The options that are available in the font class are listed below:

Option	**Possible values**
family	String specifying the name of the font.
size	Positive int for size in points, negtive int for size on pixels.
weight	One of tkFont.BOLD or tkFont.NORMAL
slant	One of tkFont.ITALIC or tkFont.NORMAL
underline	1 or 0
overstrike	1 or 0

These Font objects have two major advantages over the tuple font descriptors. First of all, it is possible to use the `config()` method of the `Font` class to modify only one property of a font. Using tuples, it is necessary to redescribe the whole font. This can become useful when responding to mouse movements for example, where a particular a font could alternate between normal and bold, keeping every other property fixed.

Also, these fonts can be assigned to variables, which make it possible to assign the same font to multiple objects. In the eventuality that the font is modifed using `config()`, then all widgets sharing the font will reflect the change simultaneously.

Here is a small example which demonstrates how to work with Font objects:

Listing 24: A tkFont.Font Example

```
from Tkinter import *
import tkFont

root = Tk()
myfont = tkFont.Font(family="arial", size=20, weight=tkFont.BOLD)
b1 = Button(root, text="Button 1", font=myfont)
b1.pack()
b2 = Button(root, text="Button 2", font=myfont)
b2.pack()
myfont.config(slant=tkFont.ITALIC)
root.mainloop()
```

This example is much more interesting when coded interactively, so that it is possible to see both buttons reflect the change in the font.

5.3 Tkinter Variables

Tkinter has a special `Variable` class that all other variable subclasses inherit from. Various Tkinter widgets have options that rely on the `Variable` class, such as the “variable” option for Radiobuttons, the “textvariable” option for the Label, etc.

5.3.1 The Need for New Variables

When the Tkinter variables are used for simple tasks, they do not behave differently than the regular Python variables, apart from the fact that their values have to be handled through method calls, such as `get()` and `set()`. The choice behind the implementation of this master variable class is therefore not obvious. However, by taking a closer look at what the Variable class offers, the motivation behind it becomes much more apparent. Tkinter variables provide, in true Tkinter style, the possibility to add a callback which will be executed when read and write operations are performed on the variable, and also when the value associated with the variable is undefined. This allows, for example, to bind a `StringVar()` instance to a `Label` widget, which then automatically updates its caption when the value of the variable changes.

5.3.2 Using Tkinter Variables

Tkinter provides four descendants of the `Variable` class:

- StringVar
- IntVar
- DoubleVar
- BooleanVar

The `Variable` class provides the following methods:

Method	Description
Private Methods	
`__init__(self, master=None)`	Constructor (an exception is raised if the master is None, since the constructor calls master.tk)
`__del__(self)`	Destructor
`__str__(self)`	Returns the name of the variable
Public Methods	
`set(self,value)`	Assigns `value` to the variable
`trace_variable(self, mode, callback)`	Assigns a callback to the variable. Mode must be one of ”r”, ”w”, or ”u”. Depending on the value of mode, callback is executed when a value is read from the variable, written to the variable or when its value becomes undefined, respectively. This method returns the name of the callback, which is needed to delete it.
`trace_vdelete(self, mode, cbname)`	Deletes the callback with the specified name and mode.
`trace_vinfo(self)`	Returns a list of all of the callback names associated with the variable and their respective modes (each mode - callback name pair is stored as a tuple).

The Variable class does not implement the get( ) method. It is only the base class for all variables, and is not intended for direct use. Instead, each descendant of the Variable class implements its specific get( ) method.

Notes:

- Tkinter variables have a __del__ method which frees their associated memory when they are no longer referenced. Therefore, one should always keep a reference to a Tkinter variable instance as a global reference or a instance variable. While this is almost always the case, it might make a difference in some very specific situations.
- It should be noted that variable options only take parameters which are instances of the `Variable` class (or one of its subclasses). If a regular Python variable is passed as parameter, then it will never receive the appropriate value. That is, the store and retrieve calls will silently fail. Moreover, it must be mentioned that the `set()` method does not do type or range checking on its parameter value.

It simply stores it, even if it is not of the proper type (Python is untyped, so this is possible). Therefore, it cannot be used as is inside a try..execute block to validate data. The get( ) method, however, will fail if the value associated with the variable is erroneous. However, at this point, the previous value is lost.

6
Event Handling: bringing applications to life

Every GUI does not do much without a way to handle events that are generated by the user. In fact, most of the execution time of graphical applications is spent in the event loop.

Tkinter provides a very simple mechanism for event handling, but a nonetheless very precise and powerful one.

6.1 The Idea

Tkinter provides an easy and convenient way to handle events by allowing callback functions to be associated with any event and any widget. In addition, very precise events can be directly described and associated with actions. For example, one could decide to only respond two two specific key press events, pressing the tabulation key or the carriage return key. While in most other languages / GUI toolkits this would require responding to the “Key Press” event and checking if the event that occured matches one of the keys we are interested in, and then executing some code depending on the result we obtained, Tkinter does it differently. It allows to bind distinct callbacks to distinct key press events, in our case one event (and one callback) for the tabulation key and one event for the carriage return key. Thus, Tkinter events do not necessarily correspond to the common event types, but they can be much more specific. Needless to say, this is a great advantage, and allows for a much cleaner implementation.

6.2 Event Types and Properties

While event descriptors can be very specific, Tkinter also supports more general events. Currently, the following event types are available, and can be classified into tree major categories:

Keyboard events : KeyPress, KeyRelease

Mouse events : ButtonPress, ButtonRelease, Motion, Enter, Leave, MouseWheel

Window events : Visibility, Unmap, Map, Expose, FocusIn, FocusOut, Circulate, Colourmap, Gravity, Reparent, Property, Destroy, Activate, Deactivate

Each callback that is bound to an event reveices and an instance of the Event class as parameter when it is invoked. The Event class defines the following properties:

Property	**Description**
char	pressed character (as a char)(KeyPress, KeyRelease only)
delta	delta of wheel movement (MouseWheel)
focus	boolean which indicates whether the window has the focus (Enter, Leave only)
height	height of the exposed window (Configure, Expose only)
keycode	keycode of the pressed key (KeyPress, KeyRelease only)
keysym	keysym of the the event as a string (KeyPress, KeyRelease only)
keysym_num	keysym of the event as a number (KeyPress, KeyRelease only)
num	number of the mouse button pressed (1=LEFT, 2=CENTER, 3=RIGHT, etc.) (ButtonPress, ButtonRelease only)
serial	serial number of the event
state	state of the event as a number (ButtonPress, ButtonRelease, Enter, KeyPress, KeyRelease, Leave, Motion only) or string indicating the source of the Visibility event (one of “VisibilityUnobscured”, “VisibilityPartiallyObscured”, and “VisibilityFullyObscured”
time	time at which the event occurred. Under Microsoft Windows, this is the value returned by the GetTickCount( ) API function.

Continued on next page

	Continued from previous page
Property	**Description**
type	type of the event as a number
x	x-position of the mouse relative to the widget
x_root	x-position of the mouse on the screen relative to the root (ButtonPress, ButtonRelease, KeyPress, KeyRelease, Motion only)
y	y-position of the mouse relative to the widget
y_root	y-position of the mouse on the screen relative to the root (ButtonPress, ButtonRelease, KeyPress, KeyRelease, Motion only)
widget	widget for which the event occurred
width	width of the exposed window (Configure, Expose only)

6.3 Event Descriptors

In Tkinter, all event handling requires a string description of the event to be bound. Each of these strings can be divided into three sections, and is usually enclosed in angular brackets. The general format is as follows:

```
<Modifier - Type - Qualifier>
```

Not all three sections are required in order for an event descriptor string to be valid. In fact, event descriptors only rarely include the three sections. The meaning and the possible values for each section will de discussed next.

1. **Type**: The type is often the only required section in an event descriptor. It specifies the kind or class of the event. The values that it can take are as follows:

Value	Triggered when ...
Key	a key is pressed while the widget has the focus
KeyRelease	a key is pressed while the widget has the focus
Button, ButtonPress	a mouse button is pressed on the widget
ButtonRelease	a mouse button is released on the widget
Motion	the mouse is moved over the widget
Enter	the mouse enters a widget
Leave	the mouse leaves a widget
FocusIn	a widget gets the focus
FocusOut	a widget loses the focus
Expose	a widget or part of a widget becomes visible
Visibility	the visibility of a widget or part of a widget changes (the state value is one of “VisibilityUnobscured”, “VisibilityPartiallyObscured”, and “VisibilityFullyObscured”
Destroy	a widget is destroyed
MouseWheel	the mouse wheel is activated over a widget

Other event types include Activate, Circulate, Colormap, Configure, Deactivate, Gravity, Map, Reparant and Unmap.

2. **Modifier**: The modifier section is optional. If present, it is used to make the event description more precise by requiring that certain keys or mouse buttons should be down while for the event to occur. It can also be used to specify that a particular event has to occur multiple times before its associated callback is executed. There can be more than one modifier specified for an event descriptor, provided that they are seperated by spaces or dashes. Whenever modifiers are present, events must *at least* match all of the specified modifiers in order for their associated callbacks to be executed. That is, if extra modifiers are present, the callback will still be executed. The values that it can take are as follows:

Value	Description
Any	Event can be triggered in any mode
Alt	Alt key must be down for event to be triggered.
Lock	Caps Lock must be enabled for event to fire.
Control	Control key must be down for event to be triggered.
Meta, M	Meta key must be down for event to be triggered.
Mod1, M1	Mod1 key should be help down for event to be triggered.
Mod2, M2	Mod2 key should be help down for event to be triggered.
⋮	
Mod5, M5	Mod5 key should be help down for event to be triggered.
Shift	Shift key must be down for event to be triggered.
Button1, B1	Mouse button 1 should be help down for event to be triggered.
Button2, B2	Mouse button 2 should be help down for event to be triggered.
⋮	
Button5, B5	Mouse button 5 should be help down for event to be triggered.
Double	The event should occur twice (in a short time period) before its callback is executed.
Triple	The event should occur three times (in a short time period) before its callback is executed.

3. **Qualifier** The qualifier can be any integer from 1 to 5, in which case it specifies a mouse button. Mouse buttons are numbered from 1 to 5 starting from the left button (left = 1, center = 2, right = 3, etc.).It can also be a keysym[1]. Keysyms are names attributed to specific keys, such as “backslash” or “backspace”.Whenever a qualifier is specified, the type is not strictly required. It is omitted, then the default type will be KeyPress for keysym qualifiers and ButtonPress for mouse button qualifiers.

The event descriptor “$< keysym >$” is equivalent to “*keysym*” (without the angular brackets), provided that *keysym* is not a space or an angular bracket.

Some example of common event descriptors are:

[1] The current version of Tk defines more than 900 keysyms

`<Any-Enter>`	Triggered when the mouse enters a widget
`<Button-1>, <1>`	Triggered when a left click is done in a widget
`<B1-Motion>`	Triggered when the mouse is dragged over the widget with the left mouse button being held down.
`<Double-Button-1>`	Triggered when a widget is double-clicked with the left mouse button
`<Key-A>, <KeyPress-A>, <A>, A`	Triggered when the key producing the letter A (caps) is pressed.

6.4 Binding Callbacks to Events

For the application to be able to respond to events, methods or functions have to be associated with events, so that they are called whenever the events occur. These methods and functions are referred to as *callbacks*. *Binding* is the process by which a callback is associated to an event by means of an event descriptor.

There are 3 methods that can be used to bind events to widgets in Tkinter:

1. The `bind()` method, which can be called on any widget, and in particular a Toplevel widget (Tkinter makes a difference between Toplevel widgets and other widgets)
2. The `bind_class()` method, which binds events to a particular widget class. This is used internally in order to provide standard bindings for Tkinter widgets. This function should usually be avoided, as it can often be replaced by implementing a descendant of the particular widget, and by binding it to the desired callback.
3. The `bind_all()` method, which binds events for the whole application.

The `bind()` method is declared as follows:

```
bind(self, sequence=None, func=None, add=None)
```

where `sequence` is an event descriptor (as a string), `func` is the name of the function to be associated with the event (ie the callback), and `add` is a boolean which specifies that `func` should be called in addition to the current callback for this event instead of replacing it. This method returns a function ID which can be used with `unbind` to remove a particular binding. The `unbind` method is defined as follows:

```
unbind(self, sequence, funcid=None)
```

Similarly, the `bind_class()` method is defined as follows:

```
bind_class(self, className, sequence=None, func=None, add=None)
```

and has its associated `unbind_class` method which is defined as follows:

```
undind_class(self, className, sequence)
```

Finally, the `bind_all()` method is defined as follows:

```
bind(self, sequence=None, func=None, add=None)
```

while the `unbind_all()` method is defined as follows:

```
unbind_class(self, sequence=None)
```

The callbacks will receive an Event object as parameter, so they must include this object in their parameter list. The Event object should be the first parameter for callbacks which are not part of a class, and the second parameter for class methods (since the class reference, self, is always the first parameter).

When an event occurs in Tkinter, the event callbacks are called according to their level of specificity. The most specific event handler is always used, the others are not called (eg when Return is pressed, a callback associated with the `<Return>` event will be executed and one associated with the `<KeyPress>` event will not be called in the case where both have been bound to the same widget). Moreover, the callbacks for the 4 different levels of Tkinter's event handling will be called in sequence, starting with the widget level, then the Toplevel, the class and then the Application.

If, at a given level, the callback handling must stop, the callback should return the string "break". This will prevent any other callbacks from being executed for a given event. This is especially useful for overriding default behaviour of certain keys (eg. backspace and tab) which occur at the application level.

6.5 Unit testing and Graphical User Interfaces: Simulating Events

While unit testing is generally simple in usual, console applications, it may be more difficult to apply this concept in Tkinter applications. How to simulate the effect of the user clicking buttons and interacting with the widgets? Let's examine a simple case first. Assume that the application to be tested only consists of Buttons and one Entry widget. Also, suppose that each button is associated to a *distinct* callback, specified using the "command" option. In this case, it easy to simulate a click on a Button, since a call to `btn.invoke()` would do the trick.

In the case where event binding has taken place, all is not lost. Generating "fake" events in Tkinter is relatively easy. The Event class acts as some kind of record (or struct, for those with a C background) which holds all of its information within its fields. So, by creating a new instance of the event class and specifying the values of the required fields, it is possible to invoke the methods with a reference to this event instance as parameter, thus simulating Tkinter's behaviour.

There is still one minor problem: usually, Tkinter programs are implemented as classes, and initialization is done in the `__init__` method. So it is *not* sufficient to import the module for the Tkinter application we which to test, we must also have an instance of its primary class. This is in fact very easy to achieve. First, observe that if a Tkinter does not call mainloop() at some point, the root window that it creates will not show up. So, by including all calls to root.mainloop() (and usually root = Tk() and

app = MyApp(root) too) inside a conditional of the form `if __name__ == '__main__'`, we are guaranteed that we can safely create an instance of the MyApp class without having to worry about it waiting for events indefinitely.

Here is an example from “calctest.py” illustrating this:

```
from Tkinter import *
import calc
import unittest

calcinst = calc.App(Tk())

class DigitTest(unittest.TestCase):
    def testDigBtns(self):
        """C + num ==> num for all num"""
        self.DigitBtns = calcinst.btnDigits
        for btn in self.DigitBtns:
            calcinst.btnCClick()
            # Here, invoke() does not work since
            # the callback is bound to an event, and was not
            # specified using the "command" option
            e = Event()
            e.widget = btn
            calc.btnDigitClick(e)
            self.assertEqual(calc.display.get(), btn.cget("text"))

if __name__ == '__main__':
    unittest.main()
```

A
Tkinter color names

Color Name	Red	Green	Blue
alice blue	240	248	255
AliceBlue	240	248	255
antique white	250	235	215
AntiqueWhite	250	235	215
AntiqueWhite1	255	239	219
AntiqueWhite2	238	223	204
AntiqueWhite3	205	192	176
AntiqueWhite4	139	131	120
aquamarine	127	255	212
aquamarine1	127	255	212
aquamarine2	118	238	198
aquamarine3	102	205	170
aquamarine4	69	139	116
azure	240	255	255
azure1	240	255	255
azure2	224	238	238
azure3	193	205	205
azure4	131	139	139
beige	245	245	220
bisque	255	228	196
bisque1	255	228	196

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
bisque2	238	213	183
bisque3	205	183	158
bisque4	139	125	107
black	0	0	0
blanched almond	255	235	205
BlanchedAlmond	255	235	205
blue	0	0	255
blue violet	138	43	226
blue1	0	0	255
blue2	0	0	238
blue3	0	0	205
blue4	0	0	139
BlueViolet	138	43	226
brown	165	42	42
brown1	255	64	64
brown2	238	59	59
brown3	205	51	51
brown4	139	35	35
burlywood	222	184	135
burlywood1	255	211	155
burlywood2	238	197	145
burlywood3	205	170	125
burlywood4	139	115	85
cadet blue	95	158	160
CadetBlue	95	158	160
CadetBlue1	152	245	255
CadetBlue2	142	229	238
CadetBlue3	122	197	205
CadetBlue4	83	134	139
chartreuse	127	255	0
chartreuse1	127	255	0
chartreuse2	118	238	0
chartreuse3	102	205	0
chartreuse4	69	139	0
chocolate	210	105	30
chocolate1	255	127	36
chocolate2	238	118	33

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
chocolate3	205	102	29
chocolate4	139	69	19
coral	255	127	80
coral1	255	114	86
coral2	238	106	80
coral3	205	91	69
coral4	139	62	47
cornflower blue	100	149	237
CornflowerBlue	100	149	237
cornsilk	255	248	220
cornsilk1	255	248	220
cornsilk2	238	232	205
cornsilk3	205	200	177
cornsilk4	139	136	120
cyan	0	255	255
cyan1	0	255	255
cyan2	0	238	238
cyan3	0	205	205
cyan4	0	139	139
dark blue	0	0	139
dark cyan	0	139	139
dark goldenrod	184	134	11
dark gray	169	169	169
dark green	0	100	0
dark grey	169	169	169
dark khaki	189	183	107
dark magenta	139	0	139
dark olive green	85	107	47
dark orange	255	140	0
dark orchid	153	50	204
dark red	139	0	0
dark salmon	233	150	122
dark sea green	143	188	143
dark slate blue	72	61	139
dark slate gray	47	79	79
dark slate grey	47	79	79
dark turquoise	0	206	209

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
dark violet	148	0	211
DarkBlue	0	0	139
DarkCyan	0	139	139
DarkGoldenrod	184	134	11
DarkGoldenrod1	255	185	15
DarkGoldenrod2	238	173	14
DarkGoldenrod3	205	149	12
DarkGoldenrod4	139	101	8
DarkGray	169	169	169
DarkGreen	0	100	0
DarkGrey	169	169	169
DarkKhaki	189	183	107
DarkMagenta	139	0	139
DarkOliveGreen	85	107	47
DarkOliveGreen1	202	255	112
DarkOliveGreen2	188	238	104
DarkOliveGreen3	162	205	90
DarkOliveGreen4	110	139	61
DarkOrange	255	140	0
DarkOrange1	255	127	0
DarkOrange2	238	118	0
DarkOrange3	205	102	0
DarkOrange4	139	69	0
DarkOrchid	153	50	204
DarkOrchid1	191	62	255
DarkOrchid2	178	58	238
DarkOrchid3	154	50	205
DarkOrchid4	104	34	139
DarkRed	139	0	0
DarkSalmon	233	150	122
DarkSeaGreen	143	188	143
DarkSeaGreen1	193	255	193
DarkSeaGreen2	180	238	180
DarkSeaGreen3	155	205	155
DarkSeaGreen4	105	139	105
DarkSlateBlue	72	61	139
DarkSlateGray	47	79	79

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
DarkSlateGray1	151	255	255
DarkSlateGray2	141	238	238
DarkSlateGray3	121	205	205
DarkSlateGray4	82	139	139
DarkSlateGrey	47	79	79
DarkTurquoise	0	206	209
DarkViolet	148	0	211
deep pink	255	20	147
deep sky blue	0	191	255
DeepPink	255	20	147
DeepPink1	255	20	147
DeepPink2	238	18	137
DeepPink3	205	16	118
DeepPink4	139	10	80
DeepSkyBlue	0	191	255
DeepSkyBlue1	0	191	255
DeepSkyBlue2	0	178	238
DeepSkyBlue3	0	154	205
DeepSkyBlue4	0	104	139
dim gray	105	105	105
dim grey	105	105	105
DimGray	105	105	105
DimGrey	105	105	105
dodger blue	30	144	255
DodgerBlue	30	144	255
DodgerBlue1	30	144	255
DodgerBlue2	28	134	238
DodgerBlue3	24	116	205
DodgerBlue4	16	78	139
firebrick	178	34	34
firebrick1	255	48	48
firebrick2	238	44	44
firebrick3	205	38	38
firebrick4	139	26	26
floral white	255	250	240
FloralWhite	255	250	240
forest green	34	139	34

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
ForestGreen	34	139	34
gainsboro	220	220	220
ghost white	248	248	255
GhostWhite	248	248	255
gold	255	215	0
gold1	255	215	0
gold2	238	201	0
gold3	205	173	0
gold4	139	117	0
goldenrod	218	165	32
goldenrod1	255	193	37
goldenrod2	238	180	34
goldenrod3	205	155	29
goldenrod4	139	105	20
gray	190	190	190
gray0	0	0	0
gray1	3	3	3
gray10	26	26	26
gray100	255	255	255
gray11	28	28	28
gray12	31	31	31
gray13	33	33	33
gray14	36	36	36
gray15	38	38	38
gray16	41	41	41
gray17	43	43	43
gray18	46	46	46
gray19	48	48	48
gray2	5	5	5
gray20	51	51	51
gray21	54	54	54
gray22	56	56	56
gray23	59	59	59
gray24	61	61	61
gray25	64	64	64
gray26	66	66	66
gray27	69	69	69

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
gray28	71	71	71
gray29	74	74	74
gray3	8	8	8
gray30	77	77	77
gray31	79	79	79
gray32	82	82	82
gray33	84	84	84
gray34	87	87	87
gray35	89	89	89
gray36	92	92	92
gray37	94	94	94
gray38	97	97	97
gray39	99	99	99
gray4	10	10	10
gray40	102	102	102
gray41	105	105	105
gray42	107	107	107
gray43	110	110	110
gray44	112	112	112
gray45	115	115	115
gray46	117	117	117
gray47	120	120	120
gray48	122	122	122
gray49	125	125	125
gray5	13	13	13
gray50	127	127	127
gray51	130	130	130
gray52	133	133	133
gray53	135	135	135
gray54	138	138	138
gray55	140	140	140
gray56	143	143	143
gray57	145	145	145
gray58	148	148	148
gray59	150	150	150
gray6	15	15	15
gray60	153	153	153

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
gray61	156	156	156
gray62	158	158	158
gray63	161	161	161
gray64	163	163	163
gray65	166	166	166
gray66	168	168	168
gray67	171	171	171
gray68	173	173	173
gray69	176	176	176
gray7	18	18	18
gray70	179	179	179
gray71	181	181	181
gray72	184	184	184
gray73	186	186	186
gray74	189	189	189
gray75	191	191	191
gray76	194	194	194
gray77	196	196	196
gray78	199	199	199
gray79	201	201	201
gray8	20	20	20
gray80	204	204	204
gray81	207	207	207
gray82	209	209	209
gray83	212	212	212
gray84	214	214	214
gray85	217	217	217
gray86	219	219	219
gray87	222	222	222
gray88	224	224	224
gray89	227	227	227
gray9	23	23	23
gray90	229	229	229
gray91	232	232	232
gray92	235	235	235
gray93	237	237	237
gray94	240	240	240

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
gray95	242	242	242
gray96	245	245	245
gray97	247	247	247
gray98	250	250	250
gray99	252	252	252
green	0	255	0
green yellow	173	255	47
green1	0	255	0
green2	0	238	0
green3	0	205	0
green4	0	139	0
GreenYellow	173	255	47
grey	190	190	190
grey0	0	0	0
grey1	3	3	3
grey10	26	26	26
grey100	255	255	255
grey11	28	28	28
grey12	31	31	31
grey13	33	33	33
grey14	36	36	36
grey15	38	38	38
grey16	41	41	41
grey17	43	43	43
grey18	46	46	46
grey19	48	48	48
grey2	5	5	5
grey20	51	51	51
grey21	54	54	54
grey22	56	56	56
grey23	59	59	59
grey24	61	61	61
grey25	64	64	64
grey26	66	66	66
grey27	69	69	69
grey28	71	71	71
grey29	74	74	74

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
grey3	8	8	8
grey30	77	77	77
grey31	79	79	79
grey32	82	82	82
grey33	84	84	84
grey34	87	87	87
grey35	89	89	89
grey36	92	92	92
grey37	94	94	94
grey38	97	97	97
grey39	99	99	99
grey4	10	10	10
grey40	102	102	102
grey41	105	105	105
grey42	107	107	107
grey43	110	110	110
grey44	112	112	112
grey45	115	115	115
grey46	117	117	117
grey47	120	120	120
grey48	122	122	122
grey49	125	125	125
grey5	13	13	13
grey50	127	127	127
grey51	130	130	130
grey52	133	133	133
grey53	135	135	135
grey54	138	138	138
grey55	140	140	140
grey56	143	143	143
grey57	145	145	145
grey58	148	148	148
grey59	150	150	150
grey6	15	15	15
grey60	153	153	153
grey61	156	156	156
grey62	158	158	158

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
grey63	161	161	161
grey64	163	163	163
grey65	166	166	166
grey66	168	168	168
grey67	171	171	171
grey68	173	173	173
grey69	176	176	176
grey7	18	18	18
grey70	179	179	179
grey71	181	181	181
grey72	184	184	184
grey73	186	186	186
grey74	189	189	189
grey75	191	191	191
grey76	194	194	194
grey77	196	196	196
grey78	199	199	199
grey79	201	201	201
grey8	20	20	20
grey80	204	204	204
grey81	207	207	207
grey82	209	209	209
grey83	212	212	212
grey84	214	214	214
grey85	217	217	217
grey86	219	219	219
grey87	222	222	222
grey88	224	224	224
grey89	227	227	227
grey9	23	23	23
grey90	229	229	229
grey91	232	232	232
grey92	235	235	235
grey93	237	237	237
grey94	240	240	240
grey95	242	242	242
grey96	245	245	245

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
grey97	247	247	247
grey98	250	250	250
grey99	252	252	252
honeydew	240	255	240
honeydew1	240	255	240
honeydew2	224	238	224
honeydew3	193	205	193
honeydew4	131	139	131
hot pink	255	105	180
HotPink	255	105	180
HotPink1	255	110	180
HotPink2	238	106	167
HotPink3	205	96	144
HotPink4	139	58	98
indian red	205	92	92
IndianRed	205	92	92
IndianRed1	255	106	106
IndianRed2	238	99	99
IndianRed3	205	85	85
IndianRed4	139	58	58
ivory	255	255	240
ivory1	255	255	240
ivory2	238	238	224
ivory3	205	205	193
ivory4	139	139	131
khaki	240	230	140
khaki1	255	246	143
khaki2	238	230	133
khaki3	205	198	115
khaki4	139	134	78
lavender	230	230	250
lavender blush	255	240	245
LavenderBlush	255	240	245
LavenderBlush1	255	240	245
LavenderBlush2	238	224	229
LavenderBlush3	205	193	197
LavenderBlush4	139	131	134

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
lawn green	124	252	0
LawnGreen	124	252	0
lemon chiffon	255	250	205
LemonChiffon	255	250	205
LemonChiffon1	255	250	205
LemonChiffon2	238	233	191
LemonChiffon3	205	201	165
LemonChiffon4	139	137	112
light blue	173	216	230
light coral	240	128	128
light cyan	224	255	255
light goldenrod	238	221	130
light goldenrod yellow	250	250	210
light gray	211	211	211
light green	144	238	144
light grey	211	211	211
light pink	255	182	193
light salmon	255	160	122
light sea green	32	178	170
light sky blue	135	206	250
light slate blue	132	112	255
light slate gray	119	136	153
light slate grey	119	136	153
light steel blue	176	196	222
light yellow	255	255	224
LightBlue	173	216	230
LightBlue1	191	239	255
LightBlue2	178	223	238
LightBlue3	154	192	205
LightBlue4	104	131	139
LightCoral	240	128	128
LightCyan	224	255	255
LightCyan1	224	255	255
LightCyan2	209	238	238
LightCyan3	180	205	205
LightCyan4	122	139	139
LightGoldenrod	238	221	130

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
LightGoldenrod1	255	236	139
LightGoldenrod2	238	220	130
LightGoldenrod3	205	190	112
LightGoldenrod4	139	129	76
LightGoldenrodYellow	250	250	210
LightGray	211	211	211
LightGreen	144	238	144
LightGrey	211	211	211
LightPink	255	182	193
LightPink1	255	174	185
LightPink2	238	162	173
LightPink3	205	140	149
LightPink4	139	95	101
LightSalmon	255	160	122
LightSalmon1	255	160	122
LightSalmon2	238	149	114
LightSalmon3	205	129	98
LightSalmon4	139	87	66
LightSeaGreen	32	178	170
LightSkyBlue	135	206	250
LightSkyBlue1	176	226	255
LightSkyBlue2	164	211	238
LightSkyBlue3	141	182	205
LightSkyBlue4	96	123	139
LightSlateBlue	132	112	255
LightSlateGray	119	136	153
LightSlateGrey	119	136	153
LightSteelBlue	176	196	222
LightSteelBlue1	202	225	255
LightSteelBlue2	188	210	238
LightSteelBlue3	162	181	205
LightSteelBlue4	110	123	139
LightYellow	255	255	224
LightYellow1	255	255	224
LightYellow2	238	238	209
LightYellow3	205	205	180
LightYellow4	139	139	122

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
lime green	50	205	50
LimeGreen	50	205	50
linen	250	240	230
magenta	255	0	255
magenta1	255	0	255
magenta2	238	0	238
magenta3	205	0	205
magenta4	139	0	139
maroon	176	48	96
maroon1	255	52	179
maroon2	238	48	167
maroon3	205	41	144
maroon4	139	28	98
medium aquamarine	102	205	170
medium blue	0	0	205
medium orchid	186	85	211
medium purple	147	112	219
medium sea green	60	179	113
medium slate blue	123	104	238
medium spring green	0	250	154
medium turquoise	72	209	204
medium violet red	199	21	133
MediumAquamarine	102	205	170
MediumBlue	0	0	205
MediumOrchid	186	85	211
MediumOrchid1	224	102	255
MediumOrchid2	209	95	238
MediumOrchid3	180	82	205
MediumOrchid4	122	55	139
MediumPurple	147	112	219
MediumPurple1	171	130	255
MediumPurple2	159	121	238
MediumPurple3	137	104	205
MediumPurple4	93	71	139
MediumSeaGreen	60	179	113
MediumSlateBlue	123	104	238
MediumSpringGreen	0	250	154

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
MediumTurquoise	72	209	204
MediumVioletRed	199	21	133
midnight blue	25	25	112
MidnightBlue	25	25	112
mint cream	245	255	250
MintCream	245	255	250
misty rose	255	228	225
MistyRose	255	228	225
MistyRose1	255	228	225
MistyRose2	238	213	210
MistyRose3	205	183	181
MistyRose4	139	125	123
moccasin	255	228	181
navajo white	255	222	173
NavajoWhite	255	222	173
NavajoWhite1	255	222	173
NavajoWhite2	238	207	161
NavajoWhite3	205	179	139
NavajoWhite4	139	121	94
navy	0	0	128
navy blue	0	0	128
NavyBlue	0	0	128
old lace	253	245	230
OldLace	253	245	230
olive drab	107	142	35
OliveDrab	107	142	35
OliveDrab1	192	255	62
OliveDrab2	179	238	58
OliveDrab3	154	205	50
OliveDrab4	105	139	34
orange	255	165	0
orange red	255	69	0
orange1	255	165	0
orange2	238	154	0
orange3	205	133	0
orange4	139	90	0
OrangeRed	255	69	0

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
OrangeRed1	255	69	0
OrangeRed2	238	64	0
OrangeRed3	205	55	0
OrangeRed4	139	37	0
orchid	218	112	214
orchid1	255	131	250
orchid2	238	122	233
orchid3	205	105	201
orchid4	139	71	137
pale goldenrod	238	232	170
pale green	152	251	152
pale turquoise	175	238	238
pale violet red	219	112	147
PaleGoldenrod	238	232	170
PaleGreen	152	251	152
PaleGreen1	154	255	154
PaleGreen2	144	238	144
PaleGreen3	124	205	124
PaleGreen4	84	139	84
PaleTurquoise	175	238	238
PaleTurquoise1	187	255	255
PaleTurquoise2	174	238	238
PaleTurquoise3	150	205	205
PaleTurquoise4	102	139	139
PaleVioletRed	219	112	147
PaleVioletRed1	255	130	171
PaleVioletRed2	238	121	159
PaleVioletRed3	205	104	137
PaleVioletRed4	139	71	93
papaya whip	255	239	213
PapayaWhip	255	239	213
peach puff	255	218	185
PeachPuff	255	218	185
PeachPuff1	255	218	185
PeachPuff2	238	203	173
PeachPuff3	205	175	149
PeachPuff4	139	119	101

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
peru	205	133	63
pink	255	192	203
pink1	255	181	197
pink2	238	169	184
pink3	205	145	158
pink4	139	99	108
plum	221	160	221
plum1	255	187	255
plum2	238	174	238
plum3	205	150	205
plum4	139	102	139
powder blue	176	224	230
PowderBlue	176	224	230
purple	160	32	240
purple1	155	48	255
purple2	145	44	238
purple3	125	38	205
purple4	85	26	139
red	255	0	0
red1	255	0	0
red2	238	0	0
red3	205	0	0
red4	139	0	0
rosy brown	188	143	143
RosyBrown	188	143	143
RosyBrown1	255	193	193
RosyBrown2	238	180	180
RosyBrown3	205	155	155
RosyBrown4	139	105	105
royal blue	65	105	225
RoyalBlue	65	105	225
RoyalBlue1	72	118	255
RoyalBlue2	67	110	238
RoyalBlue3	58	95	205
RoyalBlue4	39	64	139
saddle brown	139	69	19
SaddleBrown	139	69	19

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
salmon	250	128	114
salmon1	255	140	105
salmon2	238	130	98
salmon3	205	112	84
salmon4	139	76	57
sandy brown	244	164	96
SandyBrown	244	164	96
sea green	46	139	87
SeaGreen	46	139	87
SeaGreen1	84	255	159
SeaGreen2	78	238	148
SeaGreen3	67	205	128
SeaGreen4	46	139	87
seashell	255	245	238
seashell1	255	245	238
seashell2	238	229	222
seashell3	205	197	191
seashell4	139	134	130
sienna	160	82	45
sienna1	255	130	71
sienna2	238	121	66
sienna3	205	104	57
sienna4	139	71	38
sky blue	135	206	235
SkyBlue	135	206	235
SkyBlue1	135	206	255
SkyBlue2	126	192	238
SkyBlue3	108	166	205
SkyBlue4	74	112	139
slate blue	106	90	205
slate gray	112	128	144
slate grey	112	128	144
SlateBlue	106	90	205
SlateBlue1	131	111	255
SlateBlue2	122	103	238
SlateBlue3	105	89	205
SlateBlue4	71	60	139

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
SlateGray	112	128	144
SlateGray1	198	226	255
SlateGray2	185	211	238
SlateGray3	159	182	205
SlateGray4	108	123	139
SlateGrey	112	128	144
snow	255	250	250
snow1	255	250	250
snow2	238	233	233
snow3	205	201	201
snow4	139	137	137
spring green	0	255	127
SpringGreen	0	255	127
SpringGreen1	0	255	127
SpringGreen2	0	238	118
SpringGreen3	0	205	102
SpringGreen4	0	139	69
steel blue	70	130	180
SteelBlue	70	130	180
SteelBlue1	99	184	255
SteelBlue2	92	172	238
SteelBlue3	79	148	205
SteelBlue4	54	100	139
tan	210	180	140
tan1	255	165	79
tan2	238	154	73
tan3	205	133	63
tan4	139	90	43
thistle	216	191	216
thistle1	255	225	255
thistle2	238	210	238
thistle3	205	181	205
thistle4	139	123	139
tomato	255	99	71
tomato1	255	99	71
tomato2	238	92	66
tomato3	205	79	57

Continued on next page...

Tkinter color names - Continued

Color Name	Red	Green	Blue
tomato4	139	54	38
turquoise	64	224	208
turquoise1	0	245	255
turquoise2	0	229	238
turquoise3	0	197	205
turquoise4	0	134	139
violet	238	130	238
violet red	208	32	144
VioletRed	208	32	144
VioletRed1	255	62	150
VioletRed2	238	58	140
VioletRed3	205	50	120
VioletRed4	139	34	82
wheat	245	222	179
wheat1	255	231	186
wheat2	238	216	174
wheat3	205	186	150
wheat4	139	126	102
white	255	255	255
white smoke	245	245	245
WhiteSmoke	245	245	245
yellow	255	255	0
yellow green	154	205	50
yellow1	255	255	0
yellow2	238	238	0
yellow3	205	205	0
yellow4	139	139	0
YellowGreen	154	205	50